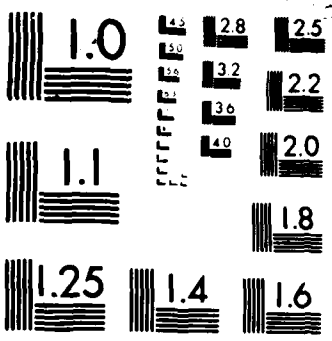
AD-A177 774 PARTITIONING PARALLEL PROGRAMS FOR MACRO-DATAFLOW(U) 1/1
STANFORD UNIV CA COMPUTER SYSTEMS LAB V SARKAR ET AL.
1986 MDA903-83-C-0335

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

Contract MDA-903-83-C-0335

Partitioning Parallel Programs for Macro-Dataflow

Vivek Sarkar and John Hennessy
Computer Systems Laboratory
Stanford University

Abstract

Partitioning techniques are necessary to execute functional programs at a coarse granularity. Fine granularity execution is inefficient on general purpose multiprocessors. There is a trade-off between parallelism and the overhead of exploiting parallelism. We present a compile-time partitioning approach to achieve this trade-off.

1. Introduction

Functional programs offer implicit parallelism at all levels. Data dependencies are their only sequencing constraints. Several parallel evaluation models exist for functional languages, e.g. dataflow [8], graph reduction [17], Concurrent Prolog [14]. These models define a granularity of parallelism at the finest level possible, e.g. instructions in dataflow, combinators in graph reduction, goals in Concurrent Prolog. The enormous scheduling and communication overhead incurred by fine grain parallelism has prompted several implementers to attempt a coarser granularity. In some implementations, the level of granularity is determined by language constructs, such as compound expressions or user-defined functions, causing the programming style to dramatically affect multiprocessor performance. We believe that the optimal granularity should be dictated by performance characteristics - specifically execution time, communication overhead and scheduling overhead. It should represent a trade-off between parallelism and the overhead of exploiting parallelism.

In this paper, we present a compile-time partitioning algorithm to partition program graphs into subgraphs that can execute in parallel. This partition provides a coarser granularity to efficiently implement parallel evaluation models on multiprocessors. For convenience, we define a *macro-actor* to be a dynamic invocation of a (static) subgraph. A macro-actor's inputs and outputs are determined by the corresponding inter-subgraph input and output edges. Our compile-time partitioning algorithm is driven by costs for execution times and communication sizes. We introduce a simple analytical model and derive an objective function, $F(\Pi)$, that defines the cost of partition Π. The partitioning algorithm attempts to build a partition with the smallest value of $F(\Pi)$.

The dataflow model is traditionally defined at the granularity of instructions or dataflow operators. With our compile-time partition, a *macro-dataflow* model can be defined at the granularity of macro-actors: each macro-actor executes sequentially, but there is parallelism among macro-actors.

A fundamental design decision in our approach is that a macro-actor be able to run to completion once all its inputs are available. This allows for non-preemptive run-time scheduling with no task-switching overhead. Cyclic dependencies are thus forbidden among macro-actors. This restriction is called the *convexity constraint* and is discussed in detail in Section 4.

A compile-time partitioner has been implemented to process program graphs in the intermediate language, IF1 [16]. IF1 represents computation as dataflow graphs, as described in Section 5. A list of target parameters (e.g. number of processors, communication and scheduling overhead) drives the partitioning for a given multiprocessor architecture. Using a front-end from SISAL [11] to IF1, we apply this system to programs written in the single-assignment language SISAL. However, our approach is applicable to any

This work has been supported by the National Science Foundation under grant # DCR8351269 and by the Defense Research Projects Agency under contract # MDA 903-83-C-0335.

Preprint: To be presented at the ACM Conference on Lisp & Functional Programming 1986

environment where a dataflow graph representation of a program can be obtained.

2. Overview of our approach

Our approach is to partition each function into subgraphs at an optimal intermediate granularity, dictated by the partition cost $F(\Pi)$. The three basic steps in this process are:

1. Cost Assignment: Traverse the program graph and assign execution time costs to nodes and communication size costs to edges.
2. Graph Partitioning: Partition each function's program graph into subgraphs.
3. Code Generation: Generate sequential code for each subgraph in the partition.

These phases are described in later sections. We begin by discussing the analytical model used to derive $F(\Pi)$, and the convexity constraint.

3. Analytical Model

We present a simple performance model for the concurrent execution of a functional program partitioned into macro-actors. Define

- P = number of processors,
- T_{seq} = sequential execution time of the program (excluding overhead),
- $T_{par}(\Pi)$ = parallel execution time of the program for partition Π (including overhead),
- $T_{crit}(\Pi)$ = critical path length of the program with partition Π; this is the parallel execution time on an unbounded number of processors,
- T_{sched} = constant overhead for scheduling a macro-actor.

For each macro-actor A executed in the program, define

- $T(A)$ = execution time of A (excluding overhead) so that $\Sigma_A T(A) = T_{seq}$,
- $T_{in}(A)$ = input communication overhead of A,
- $T_{out}(A)$ = output communication overhead of A,
- $O(A) = T_{sched} + T_{in}(A) + T_{out}(A)$ is the total overhead for macro-actor A,
- $T_{total}(\Pi) = \Sigma_A T(A) + O(A)$ is the total execution time, including overhead, over all macro-actors in the partitioned program.

Program execution proceeds by executing a ready macro-actor on a free processor. The total execution time for macro-actor A is assumed to be $O(A) + T(A)$. This analysis ignores run-time variation in the overhead term, $O(A)$, related to the load on scheduling and communication resources. Instead T_{sched} is the average scheduling overhead and $T_{in}(A)$, $T_{out}(A)$ are average communication overhead values for A's input and output communication sizes. We derive lower and upper bounds on $T_{par}(\Pi)$ to yield $F(\Pi)$, the cost function for partition Π.

For the lower bound, we have

$$T_{par}(\Pi) \geq T_{crit}(\Pi) \quad (1)$$

since the critical path length is the (optimal) parallel execution time on an unbounded number of processors. Also

$$T_{par}(\Pi) \geq T_{total}(\Pi)/P \quad (2)$$

since there are only P processors available for parallel execution. Combining (1) and (2) gives us

$$T_{par}(\Pi) \geq \max(T_{crit}(\Pi), T_{total}(\Pi)/P) \quad (3)$$

To establish an upper bound, we have to assume that the run-time scheduler will not be unnecessarily inefficient. More precisely, we assume that the scheduler always satisfies a request from a free processor if there are any macro-actors ready for execution. Many scheduling algorithms (e.g. list scheduling) have this property. Graham [7] has proved a general upper bound for the parallel execution time under these conditions. The following result is a direct consequence of his proof:

$$T_{par}(\Pi) < T_{crit}(\Pi) + T_{total}(\Pi)/P \quad (4)$$

$$\Rightarrow T_{par}(\Pi) < 2 \times \max(T_{crit}(\Pi), T_{total}(\Pi)/P) \quad (5)$$

(3) and (5) provide tight lower and upper bounds on $T_{par}(\Pi)$ that are within a constant factor of 2 from each other. To express these bounds in terms of compile-time values, we write $A \leftrightarrow S$ if macro-actor A is a dynamic invocation of subgraph S, and define:

- $f(S)$ = execution frequency of S; the number of macro-actors A with $A \leftrightarrow S$,
- $T(S) = \Sigma_{A \leftrightarrow S} T(A) / f(S)$ is the average execution time for subgraph S, so that $\Sigma_S f(S) \times T(S) = T_{seq}$,
- $O(S) = \Sigma_{A \leftrightarrow S} O(A) / f(S)$ is the average overhead for subgraph S.

Now rewrite $T_{total}(\Pi)$ as

$$\begin{aligned} T_{total}(\Pi) &= \Sigma_A T(A) + O(A) \\ &= T_{seq} + \Sigma_A O(A) \\ &= T_{seq} \times (1 + \Sigma_S f(S) \times O(S) / T_{seq}) \end{aligned}$$

Finally, define

$$F(\Pi) = \max(T_{crit}(\Pi) \times P / T_{seq}, 1 + \Sigma_S f(S) \times O(S) / T_{seq})$$

so that (3) and (5) can be combined to give

$$F(\Pi) \times (T_{seq}/P) \leq T_{par}(\Pi) < 2 \times F(\Pi) \times (T_{seq}/P) \quad (6)$$

For a given partition Π, the value of $T_{par}(\Pi)$ can vary by at

most a factor of 2. This is a bound on performance improvement due to enhancements in the run-time scheduling algorithm. However, there is large scope for performance improvement by reducing the value of $F(\Pi)$. $F(\Pi)$ can vary widely as we consider different partitions. By definition, $F(\Pi)$ will never be less than 1, so we'd like to make $F(\Pi)$ as close to 1 as possible.

$F(\Pi)$ nicely expresses the trade-off between parallelism and overhead. If the partition is too fine, the overhead term, $\Sigma f(S) \times O(S)$, will be large causing $F(\Pi)$ to be large. If the partition is too coarse, then $T_{crit}(\Pi)$ will be large due to loss of parallelism, causing $F(\Pi)$ to be large once again. $F(\Pi)$ is minimized at an optimal intermediate granularity.

The problem of finding the partition with the lowest $F(\Pi)$ is NP-complete in the strong sense. It is in NP because the value of $F(\Pi)$ can be computed in polynomial time. It is strongly NP-complete because the 3-PARTITION problem can be reduced to it. The 3-PARTITION problem [4] is to partition a set, A, of 3m elements with sizes s(a), $B/4 < s(a) < B/2$ and $\Sigma s(a) = mB$, into m disjoint sets $A_1, ..., A_m$ so that $\Sigma_{a \in A_i} s(a) = B$. This requires that each A_i contain exactly 3 elements from A. To reduce the 3-PARTITION problem to the problem to the problem of minimizing $F(\Pi)$, we build a program graph consisting of 3m nodes and no edges. Node a is given execution time s(a). We stipulate that each node is executed exactly once, so that $f(S) = 1$ for all subgraphs. Setting $T_{sched} = B$, makes $O(S) = B$ for all subgraphs, because there is no communication overhead. We can now solve the 3-PARTITION problem by finding a program partition with the smallest $F(\Pi)$, on m processors. The optimal value of $F(\Pi)$ is 2, and will only be achieved if the program partition satisfies the conditions of the 3-PARTITION problem.

Since the problem of finding the optimal partition is intractable, we have developed an efficient approximation algorithm to find a partition that's close to optimal. This algorithm is discussed in Section 7.

4. Convexity Constraint

As mentioned in the introduction, our system for compile-time partitioning and run-time scheduling is based on the premise that there are no cyclic dependencies among macro-actors. This allows a macro-actor to uninterruptedly run to completion, once all its inputs are available. The analysis in the previous section made this assumption as well. The main reason for this restriction is that cyclic dependencies between macro-actors can deteriorate performance by excessive task-switching. In our system, task-switching overhead is made explicit by considering each switch to be the execution of a new macro-actor. It requires a run-time system that can efficiently schedule new macro-actors. The macro-actors should be considered as tasks that share the same address space, rather than as separate processes.

Another advantage of this approach is that it greatly simplifies error recovery. When a failure is discovered, it is only necessary to recompute the macro-actor that was executing on the failed processor. Since its outputs only depend on its inputs, no interaction with any other macro-actors is necessary.

If we examine how this constraint on macro-actors translates to a constraint on the corresponding subgraphs of an acyclic dataflow graph, then this condition is more appropriately called the *convexity constraint*. A subgraph H of graph G is said to be *convex* [12] if any path P(a,b) where $a, b \in H$, is completely contained in H. This is analogous to convex geometrical figures that must completely contain all straight line paths between any two internal points. A *convex partition* is one in which all subgraphs are convex. It is easy to see that requiring inter-subgraph edges to be acyclic is equivalent to requiring that the partition be convex. Two trivial convex partitions of a directed acyclic graph are:

1. The partition that puts each vertex in a separate subgraph.
2. The partition that puts all the vertices in the same subgraph.

5. IF1 Program Graphs

Our compilation system operates on a graphical representation of programs, namely IF1 [16]. IF1 is an intermediate form for applicative languages. It is strongly based on the features of single-assignment languages such as SISAL [11] and VAL [1].

Compound Node	Subgraphs
Select	Selector, Alternatives
TagCase	Alternatives (for Union)
Forall	Generator, Body, Results
While, Until	Init, Test, Body, Returns

Figure 5-1: IF1 Compound Nodes

An IF1 program is a hierarchy of acyclic dataflow graphs [3]; the nodes denote operations and the edges carry data. Nodes are either *simple* or *compound*. A simple node's outputs are direct functions of its inputs. IF1 has about 50 simple nodes,

e.g. Plus, ArrayCatenate, FunctionCall. A compound node contains subgraphs and its outputs depend on the interaction between these subgraphs. Figure 5-1 lists the five compound nodes available in IF1.

Nodes have numbered ports connected by edges. An edge contains the node and port numbers of its producer and consumer. It also contains an optional type number, which is used for strongly typed languages like SISAL. Literals are special edges used for constant values. A literal has no producer - its value is given by a string. All data is carried by edges. No variables or memory locations are used. The dataflow edges in an IF1 program explicitly represent data dependencies, whereas the compound nodes and the function call node implicitly contain control dependencies. The run-time scheduler must take both kinds of dependencies into account when scheduling macro-actors.

Basic types include boolean, character, integer, real and double. Arrays, streams, records and unions are used to construct more complex types. Arrays are dynamically extendible. Nodes and edges in IF1 can have pragmas to carry additional information. We use pragmas to store profile-based frequency counts, communication and computation costs and graph partitions.

The hierarchical structure of IF1 programs is due to the fact that graphs can contain compound nodes, which themselves contain graphs. For example, Figure 5-2 shows the Quicksort program written in SISAL and Figure 5-3 shows the corresponding IF1 graph hierarchy for function Quicksort.

```
type Info = array[integer]

function Split(Data:Info returns Info,Info,Info)
    for E in Data
    returns
        array of E when E < Data[1]
        array of E when E = Data[1]
        array of E when E > Data[1]
    end for
end function

function Quicksort(Data:Info returns Info)
    if array_size(Data) > 1 then
        let
            L, Middle, R := Split(Data)
        in
            Quicksort(L)||Middle||Quicksort(R)
        end let
    else
        Data
    end if
end function
```

Figure 5-2: SISAL program for Quicksort

Figure 5-3 shows individual graphs enclosed in boxes. The graph for function Quicksort is at the top level. It contains exactly one Select compound node, which has Condition, True and False graphs. The solid edges connect nodes in a graph and represent data dependencies. Stippled lines connect a compound node to its child graphs and implicitly define data and control dependencies to generate the compound node's outputs from the outputs of the child graphs, e.g. for the Select, the output of the Condition graph is used to determine whether the True or False graph should be used to produce the Select's outputs. This graph hierarchy can be arbitrarily deep, reflecting the nesting of compound nodes.

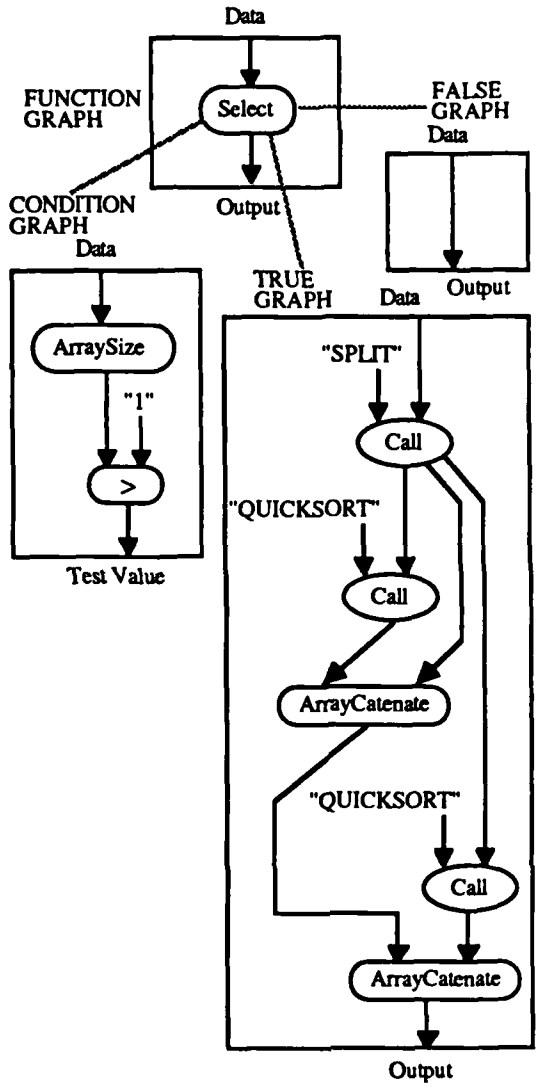


Figure 5-3: IF1 graph hierarchy for function Quicksort

6. Cost Assignment

The first step in compile-time partitioning is to estimate computation and communication costs in the program. Communication costs are determined by examining the data

type of an edge and assessing its size in an appropriate unit (e.g. bytes). Estimation of node execution times is more difficult and is undecidable in general. The unknown parameters are:

- The dynamic frequency distribution of subgraphs in a compound node (e.g. number of iterations for a While Body, probability distribution of Alternatives in a Select)
- Array size for nodes that operate on entire arrays. This parameter is also used to determine the array's communication size.
- Recursion depth for recursive function calls.

Average node execution times are determined by using average values for these frequency parameters. These frequency values can be estimated using simple rules of thumb, can be provided by the programmer through pragmas, or can be derived from profile information. Our current implementation uses profile data.

Given these parameters, it is a straightforward task to compute the cost of a node from the cost of its components via a depth-first traversal of the program graph. The cost of a function call is determined by the cost assigned to the callee. The strongly connected components (SCC's) in the call graph reveal groups of mutually recursive functions. The recursion depth estimate is used to evaluate the costs of functions in the same SCC. The reduced inter-SCC graph is acyclic and is traversed in topological order so that the callee's costs are assigned before processing the caller.

7. Partitioning Algorithm

An IF1 program is partitioned on a function by function basis. As in cost assignment, the strongly connected components in the call graph are processed in a topological order. This ensures that the callee will always be partitioned before the caller, for any non-recursive function call.

For each function, the partitioner attempts to minimize $F(\Pi) = \max(\alpha,\beta)$, where (see Section 3):

- $\alpha = 1 + \Sigma_S f(S) \times O(S) / T_{seq}$
- $\beta = T_{cnt}(\Pi) \times P / T_{seq}$

For a given program with costs and frequency information, the only parameters that can vary in α and β, when the partition Π changes, are $O(S)$ and $T_{cnt}(\Pi)$. These values are incrementally updated by the partitioner. The general structure of the partitioning algorithm is:

1. Start with the finest granularity partition that places each node in a separate subgraph.
2. Repeat steps 3 to 6 till no further merging is possible, i.e. the entire function has been included in one subgraph. Store $F(\Pi)$ for each iteration as a cost history.
3. Pick the subgraph with the largest value of $f(S)\times O(S)$ as the "best" subgraph, B, for merging. Do step 6 if B is a complete IF1 graph; otherwise do steps 4 and 5.
4. Examine the other subgraphs in B's IF1 graph as candidates for merging. For each candidate, C, compute $F(\Pi)$ for the partition obtained by merging all subgraphs in the convex hull of B and C.
5. Pick the candidate, C, with the lowest value of $F(\Pi)$ in step 4. Update the partition by merging all subgraphs in the convex hull of B and C, and go back to step 3.
6. B is a complete IF1 graph. Let P be its parent compound node. Merge B with all subgraphs in P to get a subgraph that contains all of P. There is no choice of candidates in this case, as there is only one way to merge B. Go back to step 3 after the partition has been updated.
7. Use the cost history to identify the iteration with the best partition.
8. Reconstruct the best partition.

Generally, the overhead term α decreases as the partition becomes coarser. The critical path term β may decrease initially due to reduced overhead, but will eventually increase due to loss of parallelism. Since $\alpha = 1 + \Sigma_S f(S)\times O(S)/T_{seq}$, and we want to minimize $\max(\alpha, \beta)$, the subgraph with the largest value of $f(S)\times O(S)$, i.e. the largest overhead for parallel execution, is selected for merging in step 3. All other subgraphs in the same IF1 graph are examined as candidates to merge with the selected subgraph in step 4. The candidate that yields the merged partition with lowest cost is chosen in step 5.

When merging two subgraphs, the convexity constraint requires that all subgraphs in their convex hull be merged as well. The convex hull includes those subgraphs that lie on any inter-subgraph path between the original pair. Merging all subgraphs of the convex hull guarantees that the merged partition will remain convex.

Subgraphs are merged till the end, even when $F(\Pi)$ increases, to avoid being trapped in a local minimum of $F(\Pi)$ (see Figure 7-1). The partition chosen by the algorithm has the smallest $F(\Pi)$ over all iterations, though not necessarily the optimal value. The extra number of merging iterations beyond the first minimum does not alter the worst case execution time of the algorithm.

As explained in Section 5, an IF1 function is a hierarchy of dataflow graphs, due to the presence of compound nodes. In the initial partition, all compound nodes are expanded and all the partition's subgraphs are at the lowest level. Merging continues at the same level till a subgraph chosen for merging is the entire graph of a compound node. This subgraph will then be merged with the compound node's other graphs to become a subgraph containing the compound node in the

parent graph (step 6).

A function call may contain parallelism due to the callee's partition, or may execute sequentially. The decision to parallelize or sequentialize a function call is automatically made by the partitioning algorithm. The execution time and overhead cost of a function call are initialized to the values obtained from the callee's partition. If a function call remains in a subgraph by itself, then it will run in parallel. This creates macro-actors for the callee's subgraphs at run-time. If the call is merged with other nodes, then the call will be executed sequentially and the sequential execution time and overhead are used in computing the costs and overhead of its subgraph. Generally, calls to small functions execute sequentially and calls to large functions are parallelized. Other factors may also play a role. For instance, a call with a large communication overhead for the input parameters may be better executed sequentially. In another case, it may be more efficient to merge a low frequency call with other nodes, while keeping a high frequency call, to the same function, in a subgraph by itself. Executable code for both sequential and parallel versions must be available at run-time, if a function is called both sequentially and in parallel.

In-line expansion of function calls can provide more flexibility in their partitioning. The major constraint is code size, which grows exponentially in the worst case. The IF1 system has a function integration program that serves as an optional pre-pass to our partitioner. The preceding discussion of partitioning function calls only applies to calls that were not expanded in-line.

The target multiprocessor parameters used by the partitioning algorithm to compute $F(\Pi)$ are:

- Number of processors, P.
- Scheduling overhead for a macro-actor, T_{sched}.
- Input and output communication overhead functions, $T_{in}(S)$ and $T_{out}(S)$.

The overhead terms are used to compute $O(S) = T_{sched} + T_{in}(S) + T_{out}(S)$. So far, we have used communication overhead functions of the form K × (Communication Size), where K is a communication factor that converts input/output communication size to execution time units.

Figure 7-1 shows the variation in $F(\Pi)$ with the number of merging steps, while partitioning a function from the SIMPLE benchmark. The target parameters used were:

- Number of processors, P = 10.
- Scheduling overhead, T_{sched} = 100 cycles.
- Communication overhead factor, K = 1 cycle per byte.

The function itself had $T_{seq} = 1.1 \times 10^5$ cycles.

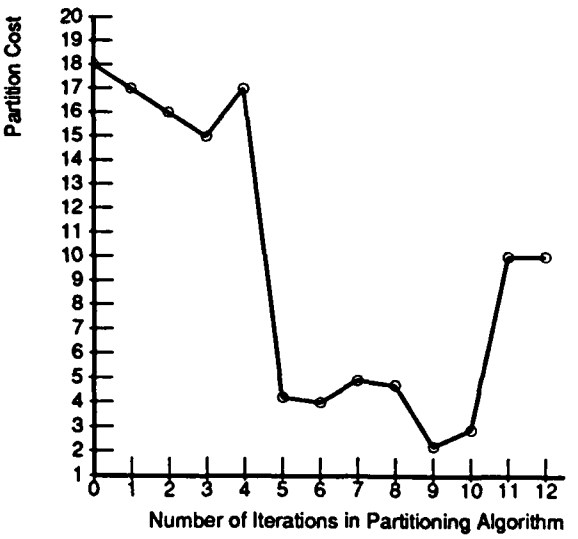


Figure 7-1: Sample variation in $F(\Pi)$

A rudimentary worst case execution time analysis for this algorithm now follows. Define:

- M = the largest number of nodes in a single IF1 graph,
- E = the largest number of edges in a single IF1 graph (note that $E = O(M^2)$),
- N = the total number of nodes over all IF1 graphs in the function,
- L = number of levels in the function's graph hierarchy.

Realistic benchmark programs show values of N and M in the ranges 500-2000 and 20-50 respectively. This suggests an $M = \sqrt{N}$ relation between M and N, which is justified when the number of graphs is comparable to the size of each graph. We also assume that the graph hierarchy is reasonably balanced so that $L = O(\log N)$.

The total number of merge iterations is O(N), since at most N-1 merges can be performed before the entire function is included in a single subgraph. A heap indexed by $f(S) \times O(S)$ is used in step 3 to efficiently pick the "best" subgraph, B. The total execution time for O(N) insertions and deletions in the heap is O(N log N).

There are O(M) candidates for the second subgraph in step 4. The complete inter-subgraph path relation (transitive closure of inter-subgraph edges) is first computed in O(M(M+E)) time. For each of the O(M) candidate subgraphs,

1. The convex hull with B is computed in O(M) time using the path relation.
2. The new overhead cost (new value of α) is computed

in O(M) time, by evaluating f(S)×O(S) for the convex hull.

3. The critical path length of a single graph can be computed in O(M+E) time. To recompute $T_{crit}(\Pi)$, the critical path length of all enclosing graphs may need to be recomputed, taking O(L(M+E)) time in the worst case.

Putting it all together, it takes O(ML(M+E)) time to pick the best candidate in step 4. Updating the partition in step 5 just takes O(L(M+E)) time.

Step 6 takes O(L(M+E)) time, since there is only one way to merge B. Only the values of α and β have to be updated. So the entire algorithm takes O(NlogN + NML(M+E)) time. Assuming $M = O(\sqrt{N})$, $E = O(M^2)$ and $L = O(\log N)$ makes this an $O(N^{2.5} \log N)$ algorithm.

8. Code Generation Issues

Each node in the IF1 program produced by the partitioner is annotated with a pragma value to indicate its subgraph. An entire subgraph is compiled to sequential code. The partitioner imposes no restrictions on the ordering of nodes within a subgraph.

The IF1 program graph representation is well suited to compile-time partitioning. However, generation of sequential machine code is more complicated than from traditional, sequential intermediate languages. It is imperative to avoid unnecessary copying when an update-in-place is possible. This effectively coalesces data on input and output edges to be the same "variable". A few research projects are under way to address this problem. The SISAL [11] project includes code generation from IF1 for the VAX 780 and Cray-2 architectures. A project is under way at Stanford to translate SAL [2] graphs (similar to IF1) to U-code [15]. Our partitioner will benefit from all advances in this field, as sequential code generation and optimization techniques can be applied to the code within a subgraph.

9. Preliminary results

The partitioning algorithm described in the Section 7 has been implemented to partition IF1 [16] program graphs. A SISAL [11] to IF1 front-end allows us to test the performance of our partitioner on programs written in the single-assignment language, SISAL. We have instrumented the Livermore IF1 interpreter to obtain statistics for a multiprocessor simulation including scheduling and communication overhead.

The simulator was carefully designed to accurately represent a multiprocessor execution. Macro-actors are assigned to processors in a breadth-first evaluation of the program. Execution time is accumulated by adding the costs of all simple nodes executed within a macro-actor. Communication sizes are derived from actual run-time values – this is particularly important for dynamic arrays. A target multiprocessor parameter is used to convert communication size (in bytes) to overhead time (in machine cycles). The scheduling overhead for a macro-actor (defined as T_{sched} in Section 3) is also a parameter of the target multiprocessor.

Figure 9-1 on the next page shows the speed-up obtained for SIMPLE, a benchmark program for computational fluid dynamics and heat flow [6]. The basic data structure is a two dimensional mesh covering the problem domain. Each iteration of the program's outer loop represents one time step, which consists of a hydrodynamics pass and a heat conduction pass. Our results are for a 10×10 mesh and a single time step. We expect to see better speed-up for a larger mesh size.

The curves in Figure 9-1 illustrate the match between a compile-time partition and the corresponding target multiprocessor parameters. Note that both axes are plotted on a logarithmic scale. The measurements were taken for two sets of parameters:

1. Low overhead - T_{sched} = 10 cycles and zero communication overhead.
2. High overhead - T_{sched} = 100 cycles and K = 1 cycle per byte.

The four curves show all combinations of the two partitions simulated on the two targets. Naturally, the low overhead target curves show a better speed-up than the high overhead target curves. But, for a given target, the partition that was generated for it performed better than the other partition. This is more significant in the presence of high overhead.

10. Related Work

The general problem of determining the optimal granularity of program decomposition has been addressed in other work. Gaudiot and Ercegovac [5] present a mean-value model of *variable resolution dataflow*. They illustrate the phenomenon of an optimal resolution that minimizes parallel execution time. Since it is a mean-value analysis, it does not apply to a particular program with a given assignment of execution times and communication sizes. Instead it describes the average performance of all programs, for a given macro-actor size. Also, they do not address the compiler issue of actually partitioning the program to achieve a desired resolution.

Hudak and Goldberg [10] introduce *serial combinators* to achieve an "optimal" granularity in graph reduction. Our work is similar in spirit to theirs. An important difference is that their serial combinators require the facility of process suspension and re-activation. As explained in Section 4, we

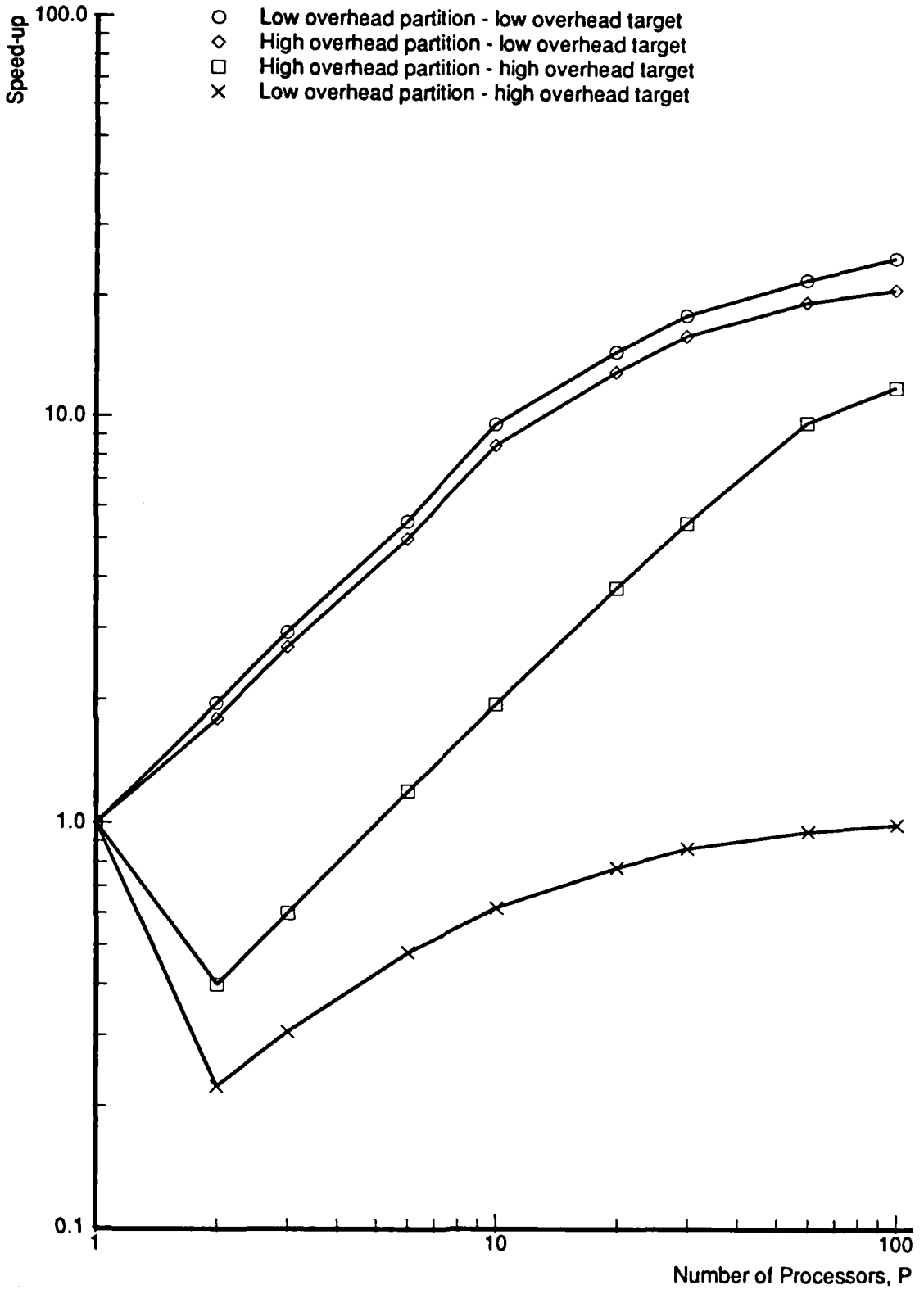


Figure 9-1: Speed-up vs. Number of processors for SIMPLE

ensure that our macro-actors can run to completion once they've been scheduled. No preemption is necessary during a macro-actor's execution. Another difference is that the serial combinators described in [10] are restricted to not contain any concurrent substructure. In our case, macro-actors may contain compound expressions and function calls even though these computations have potential parallelism. The boundary is determined entirely by costs for communication overhead and execution time.

In the Stardust system [9], both partitioning and scheduling are performed at run-time. Functions are annotated with integer valued expressions that compute execution time estimates at run-time (e.g. N log N for Quicksort). Partitioning is based on a maximum size, causing expressions with larger execution time estimates to be decomposed. The advantage of run-time partitioning is in the use of more accurate execution time estimates determined by input data at run-time. The major disadvantage is that partitioning now becomes an extra overhead in multiprocessor performance. Also, the total overhead of run-time partitioning is a function of the program's dynamic execution time, rather than its static code size. Both these factors make it mandatory for the partitioning algorithm to be very simple. Stardust's approach of a maximum size avoids sequentialization at a coarse granularity, but can incur a large overhead due to fine granularity execution.

In [13], we present an approach to compile-time scheduling intended for applications with fairly predictable run-time behavior. An IF1 program is partitioned into P sequential threads for P processors. There is no overhead due to run-time scheduling; all inter-processor synchronization and communication is directly compiled in the code. The compile-time scheduling approach operates on the same IF1 program graph representation, annotated with costs, that was described in this paper. We expect compile-time scheduling to be effective for a smaller class of programs than compile-time partitioning and run-time scheduling, but to be more efficient for that class.

11. Conclusions

We have demonstrated that the problem of partitioning functional programs at an "optimal" granularity can be solved at compile-time. Our approach is practical and has been implemented to process IF1 program graphs.

The partitioner does not assume any particular multiprocessor architecture. Instead, it is driven by a list of parameters that describe the target multiprocessor.

The convexity constraint is an important design decision for efficient run-time scheduling and improved error recovery.

Our lower and upper bound analysis of parallel execution time provides an objective function to evaluate a partition at compile-time.

The implementation has already been used to partition many small benchmark programs, and the simulation results are very encouraging. As more large benchmark programs become available, we will use this implementation as a basis to compare alternative architectures and their interaction with different applications.

References

1. Ackerman, W. B. & Dennis, J. B. VAL -- a value-oriented algorithmic language. Preliminary reference manual. MIT/LCS/TR-218, Laboratory for Computer Science, MIT, June, 1979.

2. Celoni, J. R. & Hennessy, J. L. SAL: A Single-Assignment Language for Parallel Algorithms. ClaSSiC-83-01, Center for Large Scale Scientific Computation, Stanford University, Sept., 1983.

3. Davis, A. L. & Keller, R. M. "Data Flow Program Graphs". *IEEE Computer 15*, (Feb. 1982).

4. Garey, M. R. and Johnson, D. S.. *COMPUTERS AND INTRACTABILITY A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco, 1979.

5. Gaudiot, J. J. & Ercegovac, M. D. Performance Analysis of a Data-flow computer with Variable Resolution Actors. Proc. 4th Int Conf Dist Comp Sys, 1984, pp. 2-9.

6. Gilbert, E. J. An Investigation of the Partitioning of Algorithms Across an MIMD Computing System. Technical Note No. 176, Computer Systems Laboratory, Stanford University, 1980.

7. Graham, R. L. "Bounds on Multiprocessing Timing Anomalies". *SIAM J. Appl. Math. 17*, 2 (March 1969).

8. Gurd, J. R., Kirkham, C. C. & Watson, I. "The Manchester Prototype Dataflow Computer". *CACM 28*, 1 (Jan. 1985).

9. Hornig, D. A. *Automatic Partitioning and Scheduling on a Network of Personal Computers*. Ph.D. Th., Carnegie-Mellon University, Nov. 1984.

10. Hudak, P. & Goldberg, B. Serial Combinators: "Optimal" Grains of Parallelism. Proc. Functional Programming Languages and Computer Architecture, Nancy, France, Sept., 1985, pp. 382-399.

11. McGraw, J. *et al.* SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual, Version 1.2. M-146, LLNL, March, 1985.

12. Pfaltz, J. L.. *Computer Data Structures*. McGraw-Hill, Inc., 1977.

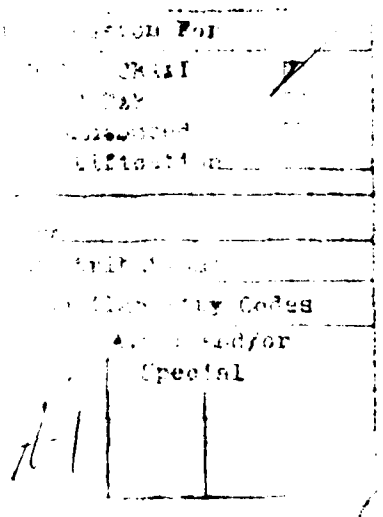
13. Sarkar, V. & Hennessy, J. L. Compile-time Partitioning and Scheduling of Parallel Programs. Proc. SIGPLAN '86 Symposium on Compiler Construction, 1986.

14. Shapiro, E. Y. A Subset of Concurrent Prolog and Its Interpreter. TR-003, The Weizmann Institute of Science, Israel, Feb., 1983.

15. Sites, R. *et al.* Machine-independent Pascal Optimizer Project: Final Report. UCSD/CS-79/038, University of California at San Diego, Nov., 1979.

16. Skedzielewski, S. & Glauert, J. IF1 -- An Intermediate Form for Applicative Languages, Version 1.0. M-170, LLNL, July, 1985.

17. Turner, D. A. "A new implementation technique for applicative languages". *Software -- Practice and Experience* 9 (1979), 31-49.



END

4-88

DTIC